Load Balancing Hybrid Programming Models for SMP Clusters and Fully Permutable Loops

Nikolaos Drosinos and Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
{ndros, nkoziris}@cslab.ece.ntua.gr

Abstract

This paper emphasizes on load balancing issues associated with hybrid programming models for the parallelization of fully permutable nested loops onto SMP clusters. Hybrid parallel programming models usually suffer from intrinsic load imbalance between threads, mainly because most existing message passing libraries generally provide limited multi-threading support, allowing only the master thread to perform inter-node message passing communication. In order to mitigate this effect, we propose a generic method for the application of static load balancing on the coarse-grain hybrid model for the appropriate distribution of the computational load to the working threads. We experimentally evaluate the efficiency of the proposed scheme against a micro-kernel benchmark, and demonstrate the potential of such load balancing schemes for the extraction of maximum performance out of hybrid parallel programs.

1 Introduction

Distributed shared memory (DSM) architectures, such as SMP clusters, have dominated the high performance computing domain by providing a reliable, cost-effective solution to both the research and the commercial communities. An immediate consequence stemming from the emersion of this new architecture is the consideration of new parallel programming models, which might exploit the underlying infrastructure more efficiently. Currently, message passing parallelization via the MPI library has become the de-facto programming approach for the development of portable code for a variety of high performance platforms. Although message passing parallel programs are generic enough, so as to be directly adapted to DSM architectures in a straightforward manner, there is an active research interest in considering alternative parallel programming models, that could be more appropriate for such platforms.

Hybrid programming models on SMP clusters resort to both message passing and shared memory access for inter- and intra-node communication, respectively, thus implementing a two-level hierarchical communication pattern. Usually, MPI is employed for the inter-node communication, while a multi-threading API, such as OpenMP, is used for the intra-node processing and synchronization. There are mainly two hybrid programming variations addressed in related work, namely the fine-grain incremental hybrid parallelization, as well as the coarse-grain SPMD-like alternative.

Fully permutable nested loop algorithms account for a large fraction of the computational intensive part of many existing scientific codes. We consider generic $N+1$-dimensional perfectly nested loops, which are parallelized across the outermost N dimensions, so as to perform sequential execution along the innermost dimension in a pipeline fashion, interleaving computation and communication phases. These algorithms impose significant communication demands, thus rendering communication-efficient parallelization schemes critical in order to obtain high performance. Moreover, the hybrid parallelization of such algorithms is a non-trivial issue, as there is a trade-off in programming complexity and parallel efficiency.

Hybrid parallelization is a popular topic in related literature, although it has admittedly delivered controversial results ([3], [7], [8], [6], [11] etc). In practice, it is still a very open subject, as the efficient use of an SMP cluster calls for appropriate scheduling methods and load balancing techniques. Most message passing libraries provide a limited multi-threading support level, allowing only the master thread to perform message passing communication. Therefore, additional load balancing must be applied so as to equalize the per tile execution times of all threads. This effect has been theoretically spotted in related literature ([12], [10], [4], [5]), but to our knowledge no generic load balancing technique has been proposed and, more importantly, evaluated.

In this paper we propose two load balancing schemes for the coarse-grain hybrid parallelization of fully permutable nested loop algorithms. The computational load associated with a tile is statically distributed among threads, based upon the estimation and modeling of basic system parameters. We distinguish between two variations of load balancing, namely both a constant and a variable scheme, depending on whether the same task distribution is applied on a global or a per process base, respectively. We emphasize on the elements of applicability and simplicity, and evaluate the efficiency of the proposed scheme against a popular micro-kernel benchmark, namely ADI integration. The experimental evaluation indicates that the variable load balancing model ensures superior performance compared to all hybrid alternatives, and further slightly outperforms the pure message model for most cases.

2 Algorithmic Model - Notation

Our algorithmic model concerns fully permutable perfectly nested loops. Such algorithms can be transformed with the aid of the tiling transformation to deliver an equivalent parallel tiled code form, that can be formally described as in Alg. 1. We assume an $N+1$-dimensional algorithm

Algorithm 1: iterative algorithm model

foracross $tile_1 \leftarrow 1$ ***to*** `H(`X_1`)` **do**
    ...
    foracross $tile_N \leftarrow 1$ ***to*** `H(`X_N`)` **do**
        for $tile_{N+1} \leftarrow 1$ ***to*** `H(`Z`)` **do**
            `Receive(`$\overrightarrow{tile}$`);`
            `Compute(`$A, \overrightarrow{tile}$`);`
            `Send(`$\overrightarrow{tile}$`);`

with an iteration space of $X_1 \times \cdots \times X_N \times Z$, which has been partitioned using a tiling transformation `H`. Z is considered the longest dimension of the iteration space, and should be brought to the innermost loop through permutation in order to simplify the generation of efficient parallel tiled code ([14]). In the above code, tiles are identified by an $N+1$-dimensional vector $(tile_1, \ldots, tile_{N+1})$. **foracross** implies parallel execution, as opposed to sequential execution (**for**). Generally, tiled code is associated with a particular tile-to-processor distribution strategy, that enforces explicit data distribution and implicit computation distribution, according to the computer-owns rule. For homogeneous platforms and fully permutable iterative algorithms, related scientific literature ([2]) has proved the optimality of the columnwise allocation of tiles to processors, as long as sequential pipelined execution along the longest dimension is assumed. Therefore, all parallel algorithms considered in this paper implement computation distribution across the N outermost dimensions, while each processor computes a sequence of tiles along the innermost $N+1$-th dimension.

In most practical cases, the data dependencies of the algorithm are of several orders of magnitude smaller compared to the iteration space dimensions. Consequently, only neighboring processes need to communicate, assuming reasonably coarse parallel granularities, taking into account that distributed memory architectures are addressed. According to the above, we only consider unitary process communication directions for our analysis, since all other non-unitary process dependencies can be satisfied according to indirect message passing techniques, such as the ones described in [13]. However, in order to preserve the communication pattern of the application, we consider a weight factor d_i for each process dependence direction i, implying that if iteration $\vec{j} = (j_1, \ldots, j_i, \ldots, j_{N+1})$ is assigned to a process $\vec{p}$, and iteration $\vec{j'} = (j_1, \ldots, j_i + d_i, \ldots, j_{N+1})$ is assigned to a different process $\vec{p'}$, $\vec{p} \neq \vec{p'}$, then data calculated at iteration $\vec{j}$ from $\vec{p}$ need to be sent to $\vec{p'}$, since they will be required for the computation of data at iteration $\vec{j'}$.

In the following, $P_1 \times \cdots \times P_N$ and $T_1 \times \cdots \times T_N$ denote the process and thread topology, respectively. $P = \prod_{i=1}^{N} P_i$ is the total number of processes, while $T = \prod_{i=1}^{N} T_i$ the total number of threads. Also, vector $\vec{p} = (p_1, \ldots, p_N)$, $0 \leq p_i \leq P_i - 1$ identifies a specific process, while $\vec{t} = (t_1, \ldots, t_N)$, $0 \leq t_i \leq T_i - 1$ refers to a particular thread. Throughout the text, we will use MPI and OpenMP notations in the proposed parallel algorithms.

3 Message Passing Parallelization

The proposed pure message passing parallelization for the algorithms described above is based on the tiling transformation. Tiling is a popular loop transformation and can be applied in the context of implementing coarse granularity in parallel programs, or even in order to exploit memory hierarchy by enforcing data locality. Tiling partitions the original iteration space of an algorithm into atomic units of execution, called tiles. Each process assumes the execution of a sequence of tiles, successive along the longest dimension of the original iteration space.

The message passing parallelization paradigm for tiled nested loops is schematically depicted in Alg. 2. Each process is identified by N-dimensional vector $\vec{p}$, while different tiles correspond to different instances of $N+1$-dimensional vector $\overrightarrow{tile}$. The N outermost coordinates of a tile specify its owner process $\vec{p}$, while the innermost coordinate $tile_{N+1}$ iterates over the set of tiles assigned to that process. z denotes the tile height along the sequential execution dimen-

Algorithm 2: pure message passing model

for $i \leftarrow 1$ ***to*** N **do**
  $tile_i = p_i$;
for $tile_{N+1} \leftarrow 1$ ***to*** $\lceil \frac{Z}{z} \rceil$ **do**
  foreach $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ **do**
    `Pack(`$\overrightarrow{dir}$,$tile_{N+1}-1$,$\vec{p}$`);`
    `MPI_Isend(`$\vec{p}+\overrightarrow{dir}$`);`
    `MPI_Irecv(`$\vec{p}-\overrightarrow{dir}$`);`
  `Compute(`$\overrightarrow{tile}$`);`
  `MPI_Waitall;`
  foreach $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ **do**
    `Unpack(`$\overrightarrow{dir}$,$tile_{N+1}+1$,$\vec{p}$`);`

sion, and determines the granularity of the achieved parallelism: higher values of z imply less frequent communication and coarser granularity, while lower values of z call for more frequent communication and lead to finer granularity.

Furthermore, advanced pipelined scheduling is adopted as follows: In each time step, a process $\vec{p} = (p_1, \ldots, p_N)$ concurrently computes a tile $(p_1, \ldots, p_N, tile_{N+1})$, receives data required for the computation of the next tile $(p_1, \ldots, p_N, tile_{N+1} + 1)$ and sends data computed at the previous tile $(p_1, \ldots, p_N, tile_{N+1} - 1)$. $\mathbb{C}_{\vec{p}}$ denotes the set of valid communication directions of process $\vec{p}$, that is, if $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ for a non-boundary process $\vec{p}$, then $\vec{p}$ needs to send data to process $\vec{p}+\overrightarrow{dir}$ and also receive data from process $\vec{p}-\overrightarrow{dir}$. $\mathbb{C}_{\vec{p}}$ is determined both by the data dependencies of the original algorithm, as well as by the selected process topology of the parallel implementation.

For the true overlapping of computation and communication, as theoretically implied by the above scheme by combining non-blocking communication primitives with the overlapping scheduling, the usage of advanced CPU offloading features is required, such as zero-copy and DMA-driven communication. Unfortunately, experimental evaluation over a standard TCP/IP based interconnection network, such as Ethernet, combined with the ch_p4 ADI-2 device of the MPICH implementation, prohibits such advanced non-blocking communication, but nevertheless the same limitations hold for our hybrid model, and are thus not likely to affect the relative performance comparison. However, this fact does complicate our theoretical analysis, since we will assume in general distinct, non-overlapped computation and communication phases, an assumption that to some extent underestimates the efficiency of the message passing communication primitives.

4 Hybrid Parallelization

The potential for hybrid parallelization is mainly limited by the multi-threading support provided by the message passing library. From the perspective of the message passing library, there are mainly five levels of multi-threading support addressed in relevant scientific literature, namely single, masteronly, funneled, serialized and multiple. Each category is a superset of all previous ones. Currently, popular non-commercial message passing libraries provide support up to the third level (funneled), allowing only the master thread to call message passing routines within the dynamic extent of multi-threaded parallel regions, while only some proprietary libraries allow for full multi-threading support (multiple thread support level). Due to this fact, most attempts for hybrid parallelization of applications, that have been proposed or implemented, are mostly restricted to the first three thread support levels.

In this Section, we propose two hybrid implementations for iterative algorithms, namely both fine- and coarse-grain hybrid parallelization. Both models implement the advanced hyperplane scheduling presented in [1].

4.1 Fine-grain Hybrid Parallelization

The fine-grain hybrid programming paradigm, also referred to as masteronly in related literature, is the most popular hybrid programming approach, although it raises a number of performance deficiencies. The popularity of the fine-grain model over a coarse-grain one is mainly attributed to its programming simplicity: in most cases, it does not require significant restructuring of the existing message passing code, and is relatively simple to implement by submitting an application to performance profiling and further parallelizing performance critical parts with the aid of multi-threading processing. Also, fine-grain parallelization is the only feasible hybrid approach for message passing libraries supporting only masteronly multi-threading.

However, the efficiency of the fine-grain hybrid model is directly associated with the fraction of the code that is incrementally parallelized, according to Amdahl's law: since message passing communication can be applied only outside of parallel regions, other threads are essentially sleeping when such communication occurs, resulting to poor CPU utilization and overall inefficient load balancing. Also, this paradigm suffers from the overhead of re-initializing the thread structures every time a parallel region is encountered, since threads are continually spawned and terminated. This thread management overhead can be substantial, especially in case of a poor implementation of the multi-threading library, and generally increases with the number of threads. Moreover, incremental loop parallelization is a very restrictive multi-threading parallelization approach for many real algorithms, where such loops either do not exist or cannot be directly enclosed by parallel regions.

Algorithm 3: fine-grain hybrid model

1 **for** $i \leftarrow 1$ ***to*** N **do**
2     $group_i = p_i$;
3 **foreach** $group_{N+1} \in \mathbb{G}_{\vec{p}}$ **do**
4   **foreach** $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ **do**
5   `Pack`($\overrightarrow{dir}$,$group_{N+1} - 1$,$\vec{p}$);
6   `MPI_Isend`($\vec{p} + \overrightarrow{dir}$);
7   `MPI_Irecv`($\vec{p} - \overrightarrow{dir}$);
8   **#pragma omp parallel**
9   **for** $i \leftarrow 1$ ***to*** N **do**
10             $tile_i = p_i T_i + t_i$;
11         $tile_{N+1} = group_{N+1}$ - $\sum_{i=1}^{N} tile_i$;
12   **if** $1 \leq tile_{N+1} \leq \lceil \frac{Z}{z} \rceil$ **then**
13   `Compute`($\overrightarrow{tile}$);
14   `MPI_Waitall`;
15   **foreach** $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ **do**
16   `Unpack`($\overrightarrow{dir}$,$group_{N+1} + 1$,$\vec{p}$);

The proposed fine-grain hybrid implementation for iterative algorithms is depicted in Alg. 3. The hyperplane scheduling is implemented as follows: Each group is identified by a $N+1$-dimensional vector $\overrightarrow{group}$, where the N outermost coordinates denote the owner process $\vec{p}$, and the innermost one iterates over the distinct time steps. $\mathbb{G}_{\vec{p}}$ corresponds to the set of time steps of process $\vec{p}$, and depends both on the process and thread topology. For each instance of vector $\overrightarrow{group}$, each thread determines a candidate tile $\overrightarrow{tile}$ for execution, and further evaluates an **if**-clause to check whether that tile is valid and should be computed at the current time step.

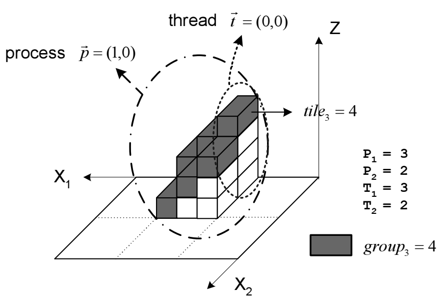


Figure 1. Hybrid parallel program for 3D algorithm and 6 processes × 6 threads

All message passing communication is performed outside of the parallel region (lines 4-7 and 14-16), while the multi-threading parallel computation occurs in lines 8-13. Note that no explicit barrier is required for thread synchronization, as this effect is implicitly achieved by exiting the multi-threaded parallel region. Note also that only the code fraction in lines 8-13 fully exploits the underlying processing infrastructure, thus effectively limiting the parallel efficiency of the algorithm. Fig. 1 clarifies some of the notation used in the hybrid algorithms.

4.2 Coarse-grain Hybrid Parallelization

According to the coarse-grain model, threads are only spawned once and their ids are used to determine their flow of execution in the SPMD-like code. Inter-node message passing communication occurs within the extent of the multi-threaded parallel region, but is completely assumed by the master thread, as dictated by the funneled thread support level. Intra-node synchronization between threads of the same SMP node is achieved with the aid of a barrier directive of the multi-threading API.

The additional promising feature of the coarse-grain approach is the potential for overlapping multi-threaded computation with message passing communication. However, due to the restriction that only the master thread is allowed to perform message passing, a naive straightforward implementation of the coarse-grain model suffers from load imbalance between the threads, if equal portions of the computational load are assigned to all threads. Therefore, additional load balancing must be applied, so that the master thread will assume a relatively smaller computational load compared to the other threads, thus equalizing the per tile execution times of all threads. Moreover, the coarse-grain model avoids the overhead of re-initializing thread structures, since threads are spawned only once, and can potentially implement more generic parallelization schemes, as opposed to its limiting fine-grain counterpart.

The pseudo-code for the coarse-grain parallelization of the fully permutable iterative algorithms is depicted in Alg. 4. Note that the inter-node communication (lines 8-12 and 16-19) is conducted by the master thread, per communication direction and per owner thread, incurring additional complexity compared to both the pure message passing and the fine-grain model. Also, note the `bal` parameter in the computation, that optionally implements load balancing between threads, as will described in Section 5.

5 Load Balancing for Hybrid Model

The hyperplane scheduling scheme enables for a more efficient load balancing between threads: Since the computations of each time step are essentially independent of

Algorithm 4: coarse-grain hybrid model

#pragma omp parallel
  for $i \leftarrow 1$ ***to*** N **do**
    $group_i = p_i$;
    $tile_i = p_i T_i + t_i$;
  foreach $group_{N+1} \in \mathbb{G}_{\vec{p}}$ **do**
    $tile_{N+1} = group_{N+1} - \sum_{i=1}^{N} tile_i$;
    #pragma omp master
      foreach $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ **do**
        for $th \leftarrow 1$ ***to*** M **do**
          `Pack(`$\vec{dir}, group_{N+1} - 1, \vec{p}, th$`);`
        `MPI_Isend(`$\vec{p} + \overrightarrow{dir}$`);`
        `MPI_Irecv(`$\vec{p} - \overrightarrow{dir}$`);`
    if $1 \leq tile_{N+1} \leq \lceil \frac{Z}{z} \rceil$ **then**
      `Compute(`$\overrightarrow{tile}$, `bal(`$\vec{p}, \vec{t}$`));`
    #pragma omp master
      `MPI_Waitall;`
      foreach $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ **do**
        for $th \leftarrow 1$ ***to*** M **do**
          `Unpack(`$\overrightarrow{dir}, group_{N+1}+1, \vec{p}, th$`);`
    #pragma omp barrier

the communication data exchanged at that step, they can be arbitrarily distributed among threads. Thus, it would be meaningful for the master thread to assume a smaller part of computational load, so that the total computation and the total communication associated with the owner process is evenly distributed among all threads.

We have implemented two alternative static load balancing schemes. The first one (*constant balancing*) requires the calculation of a constant balancing factor, which is common for all processes. For this purpose, we consider a non-boundary process, that performs communication across all N process topology dimensions, and determine the computational fraction of the master thread, that equalizes tile execution times on a per thread basis. The second scheme (*variable balancing*) requires further knowledge of the process topology, and ignores communication directions cutting the iteration space boundaries, since these do not result to actual message passing. For both schemes, the balancing factor(s) can be obtained by the following lemma:

Lemma 1 *Let $X_1 \times \cdots \times X_N \times Z$ be the iteration space of an $N+1$-dimensional iterative algorithm, that imposes data dependencies $[d_1, \ldots, 0]^T$, ..., $[0, \ldots, d_{N+1}]^T$. Let $P = P_1 \times \cdots \times P_N$ be the process topology and T the number of threads available for the parallel execution of the hybrid funneled implementation of the respective tiled*

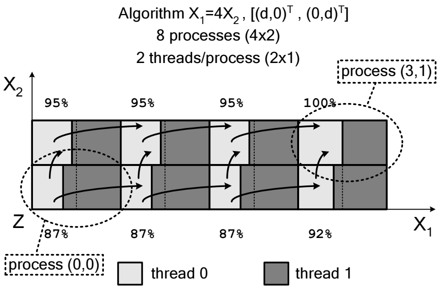


Figure 2. Variable balancing for 8 processes $\times$ 2 threads and algorithm with $X_1 = 4X_2$

algorithm. The overall completion time of the algorithm is minimal if the master thread assumes a portion $\frac{bal}{T}$ of the process's computational load, where

$$bal = 1 - \frac{T-1}{t_{comp}\left(\frac{Xz}{P}\right)} \sum_{\substack{i=1 \\ i \in \mathbb{C}_{\vec{p}}}}^{N} t_{comm}\left(\frac{d_i P_i X z}{X_i P}\right) \quad (1)$$

$t_{comp}(x)$ *The computation time required for x iterations*

$t_{comm}(x)$ *The transmission time of an x-sized message*

z *The tile height for each execution step*

$\mathbb{C}_{\vec{p}}$ *Valid communication directions of process $\vec{p}$*

X *Equal to $\prod_{i=1}^{N} X_i$*

The proof of the lemma is omitted due to space limitations. We assume the computation time t_{comp} to be a linear function of the number of iterations, that is, we assume $t_{comp}(ax) = a t_{comp}(x)$. Note that if condition $i \in \mathbb{C}_{\vec{p}}$ is evaluated separately for each process, variable balancing is enforced. If the above check is omitted, (1) delivers the constant balancing factor.

The constant balancing scheme can be applied at compile-time, since it merely requires knowledge of the underlying computational and network infrastructure, but also tends to overestimate the communication load for boundary processes. On the other hand, the variable balancing scheme can be applied only after selecting the process topology, as it uses that information to calculate a different balancing factor for each process. Fig. 2 demonstrates the variable load balancing scheme, for a dual SMP cluster. Generally, a factor bal, $0 \leq bal \leq 1$, for load balancing T threads means that the master thread assumes $\frac{bal}{T}$ of the process's computational share, while all other threads are assigned a fraction of $\frac{T-bal}{T(T-1)}$ of that share. Note that according to the variable scheme, the balancing factor is slightly smaller for non-boundary processes. However, as the active communication directions decrease for boundary processes, the balancing factor increases, so as to preserve the desirable thread load balancing.

6 Experimental Results

In order to test the efficiency of the proposed load balancing schemes, we use a micro-kernel benchmark, namely Alternating Direction Implicit integration. ADI is a stencil computation used for solving partial differential equations ([9]). Essentially, ADI is a simple three-dimensional perfectly nested loop algorithm, that imposes unitary data dependencies across all three space directions. It has an iteration space of $X_1 \times X_2 \times Z$, where Z is considered to be the longest algorithm dimension. We choose to experimentally verify the efficiency of load balancing with ADI, as it is a typical representative of tiled nested loop algorithms and complies with the target algorithmic model, that accentuates interleaving computation and communication phases. More importantly, ADI imposes communication in all three unitary directions. Consequently, all parallel implementations of ADI include a significant amount of communication, and thus facilitate the comparison of various programming models on distributed memory environments.

We use MPI as the message passing library and OpenMP as the multi-threading API. Our experimental platform is an 8-node Pentium III dual-SMP cluster interconnected with 100 Mbps FastEthernet. Each node has two Pentium III CPUs at 800 MHz, 256 MB of RAM, 16 KB of L1 I Cache, 16 KB L1 D Cache, 256 KB of L2 cache, and runs Linux with 2.4.26 kernel. For the support of OpenMP directives, we use Intel C++ compiler v.8.1 with the following optimization flags: `-O3 -mcpu=pentiumpro -openmp -static`. Finally, we use MPI implementation MPICH v.1.2.6, appropriately configured for an SMP cluster, so as to perform intra-node communication through SYS V shared memory. This version of the MPICH implementation asserts a funneled thread support level, and is thus capable of supporting all programming models discussed here. Some fine-tuning of the MPICH communication performance for our experimental platform indicated using a maximum socket buffer size of 104KB, so the respective `P4_SOCKBUFSIZE` environment variable was appropriately set to that value for all cluster nodes.

In all cases, we use 16 processes for the pure message passing experiments, and 8 processes with 2 threads per process for all hybrid programs. For the pure message passing case, an appropriate machine file is used to ensure that two MPI processes residing on the same SMP node will communicate through shared memory segments. Also, all experimental results are averaged over at least three independent executions for each case.

6.1 Load Balancing

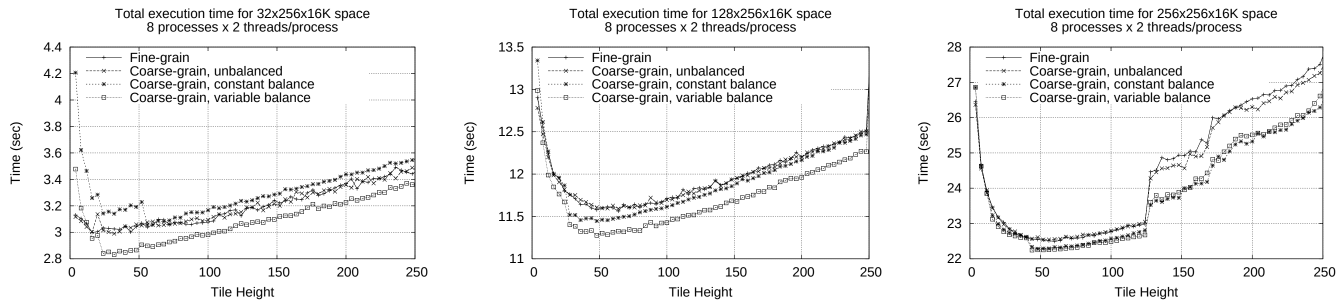
A simplistic approach is adopted in order to model the behavior of the underlying infrastructure, so as to approximate quantities t_{comp} and t_{comm} of (1). As far as t_{comp} is concerned, we assume the computational cost involved with the calculation of x iterations to be x times the average cost required for a single iteration. On the other hand, the communication cost is considered to consist of a constant start-up latency term, as well as a term proportional to the message size, that depends upon the sustained network bandwidth on application level. Formally, we define

$$t_{comp}(x) = x t_{comp}(1) \quad (2)$$

$$t_{comm}(x) = t_{startup} + \frac{x}{B_{sustained}} \quad (3)$$

Since our primary objective was preserving simplicity and applicability in the modeling of environmental parameters, we intentionally overlooked at more complex phenomena, such as cache effects or precise communication modeling. Despite having thoroughly studied the MPICH source code in order to acquire an in-depth understanding of the ch_p4 ADI-2 device, we decided not to integrate implementation-specific protocol semantics into our theoretical model in order to preserve generality and simplicity. The same holds for cache effects, which would require a memory access pattern analysis of the tiled application in respect to the memory hierarchy configuration of the underlying architecture. A more accurate representation of such hardware and software issues would probably lead to more efficient load balancing. Also, a major difficulty we encountered was modeling the TCP/IP socket communication performance and incorporating that analysis in our load balancing scheme. Assuming distinct, non-overlapping computation and communication phases and relatively high sustained network bandwidth allowed bypassing this restriction. However, this hypothesis underestimates the communication cost for short messages, which are mostly latency-bound and sustain relatively low throughput, while on the other hand it overestimates the respective cost in the case of longer messages, where DMA transfers alleviate the CPU significantly. Our main goal was providing some intuition as to the merit of these load balancing schemes, even under the most simple and straightforward implementation. For our analysis, we considered $t_{comp}(1) = 288nsec$, $t_{startup} = 107usec$ and $B_{sustained} = 100Mbps$.

The overall experimental results for ADI integration and various iteration spaces are depicted in Fig. 3. These results are normalized in respect to the fine-grain execution times. Granularity measurements for various iteration spaces are depicted in Fig. 4. Some significant performance degradations observed at certain threshold values of z can be ascribed to the transition from the eager to the rendezvous



(a) $32 \times 256 \times 16K$ (b) $128 \times 256 \times 16K$ (c) $256 \times 256 \times 16K$

Figure 4. Granularity results for ADI integration (8 dual SMP nodes)

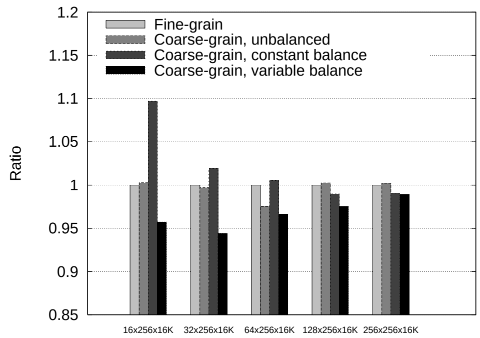


Figure 3. Comparison of hybrid models (ADI integration, various iteration spaces)

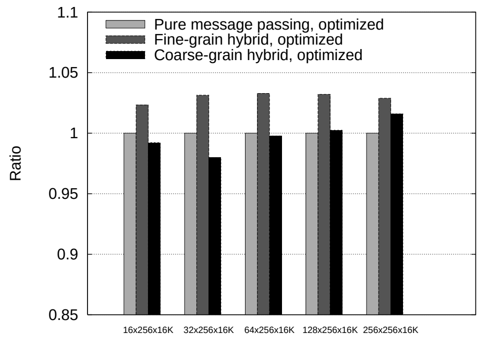


Figure 5. Overall comparison of optimized programming models on ADI integration

MPICH message protocol occurring at 128000 bytes (for instance, see Fig. 4(c) at $z = 125$).

Following conclusions can be drawn from the thorough investigation of the obtained performance measurements:

- The coarse-grain hybrid model is not always more efficient than its fine-grain counterpart. This observation reflects the fact that the poor load balancing of the simple coarse-grain model diminishes its advantages compared to the fine-grain alternative.

- When applying constant balancing, in some cases the balanced coarse-grain implementation is less effective than the unbalanced alternatives. This can be attributed both to inaccurate theoretical modeling of the system parameters for the calculation of the balancing factors, as well as to the inappropriateness of the constant balancing scheme for boundary processes.

- When applying variable balancing, the coarse-grain hybrid model was able to deliver superior performance to the fine-grain alternative in all cases. The performance improvement lies in the range of 2-8%. Furthermore, Fig. 4 reveals that variable balancing performs better than any other implementation for most granularities, thus appearing to be in almost all cases the most efficient hybrid parallelization approach.

6.2 Overall Comparison

We also performed an overall comparison of the message passing model, the fine-grain hybrid one and the variably balanced coarse-grain paradigm. We display the obtained results in Fig. 5, normalized to the pure message passing execution times. It should be noted that although this comparison may be useful towards the efficient usage of SMP clusters, it cannot be generalized beyond the chosen hardware-software combination, as it largely depends upon the comparative performance of the two programming APIs (MPICH vs OpenMP support on Intel compiler), as well as how efficiently MPICH supports multi-threaded programming.

That said, we observe that the fine-grain hybrid parallelization is always 3-8% worse in terms of performance compared to pure message passing. This observation reflects the fact that unoptimized hybrid programming suffers from serious disadvantages compared to pure message passing, and fails to exploit its structural similarities with the architecture of SMP clusters. Limiting parallelization due to the masteronly thread support level according to Amdahl's law, as well as the overhead of re-initializing the thread structures by repeatedly entering and exiting parallel regions, are the main causes for the poor performance of the fine-grain hybrid model.

On the other hand, the combination of the coarse-grain model with an efficient load balancing technique seems very promising, as it significantly improves the obtained execution times. In fact, for most cases the optimized coarse-grain hybrid model outperforms the pure message passing one, although only by a small fraction. However, some drawbacks incurred by the coarse-grain model cannot be avoided, even under all proposed optimizations, the two most fundamental of which involve the additional communication overhead when having the master thread assume all message passing, as well as the difficulties in accurately estimating the impact of the various system parameters in order to apply load balancing. As a result, the message passing and the coarse-grain model perform similarly, and only at specific cases appear slight performance differences.

7 Conclusions - Future Work

This paper discusses load balancing issues regarding hybrid parallelization of fully permutable nested loops. We propose two static load balancing schemes, namely a constant and a variable approach, that model basic system and application parameters in order to determine a suitable task distribution between threads. We compare both popular hybrid programming models, that is, fine- and coarse-grain, and experimentally evaluate the efficiency of both load balancing schemes against a micro-kernel benchmark. The evaluation demonstrates a significant performance improvement for all cases, when the variable balancing scheme is applied. We have also conducted an overall experimental comparison of the hybrid models and standard message passing parallelization. Appropriately balanced coarse-grain hybrid parallelization exhibits slightly better performance than the pure message alternative for most cases considered, although only by a small fraction.

We intend to implement a dynamic variation of the variable balancing scheme and evaluate all proposed schemes against more real applications and benchmarks. Also, we are currently investigating the performance attained in a Myrinet-interconnected SMP cluster, in order to draw more generic conclusions concerning different hardware-network combinations. Last, we are also working on improving the theoretical load balancing model, in order to more accurately model the behavior of the underlying system by also presuming simplicity and applicability of the methodology.

References

[1] M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, and N. Koziris. Pipelined Scheduling of Tiled Nested Loops onto Clusters of SMPs Using Memory Mapped Network Interfaces. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2002.

[2] P. Calland, J. Dongarra, and Y. Robert. Tiling with Limited Resources. In *Application Specific Systems, Architectures and Processors*, pages 229–238, Jun 1997.

[3] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In *Proceedings of the ACM/IEEE conference on Supercomputing*, page 12, 2000.

[4] D. G. Chavarría-Miranda and J. M. Mellor-Crummey. An Evaluation of Data-parallel Compiler Support for Line-sweep Applications. *Journal of Instruction Level Parallelism*, 5:1–29, Feb 2003.

[5] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría-Miranda. Generalized Multipartitioning of Multi-dimensional Arrays for Parallelizing Line-sweep Computations. *Journal of Parallel and Distributed Computing*, 63(9):887–911, 2003.

[6] S. Dong and G. E. Karniadakis. Dual-level Parallelism for High-order CFD Methods. *Journal of Parallel Computing*, 30(1):1–20, 2004.

[7] N. Drosinos and N. Koziris. Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters. In *Proceedings of IPDPS '04*, Apr 2004.

[8] D. S. Henty. Performance of Hybrid Message-passing and Shared-memory Parallelism for Discrete Element Modeling. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2000.

[9] G. E. Karniadakis and R. M. Kirby. *Parallel Scientific Computing in C++ and MPI : A Seamless Approach to Parallel Algorithms and their Implementation.* 2002.

[10] A. Legrand, H. Renard, Y. Robert, and F. Vivien. Mapping and Load-balancing Iterative Computations on Heterogeneous Clusters with Shared Links. *IEEE Trans. on Parallel and Distributed Systems*, 15(6):546–558, Jun 2004.

[11] R. D. Loft, S. J. Thomas, and J. M. Dennis. Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2001.

[12] R. Rabenseifner and G. Wellein. Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures. *International Journal of High Performance Computing Applications*, 17(1):49–62, 2003.

[13] P. Tang and J. Zigman. Reducing Data Communication Overhead for DOACROSS Loop Nests. In *Proceedings of the 8th International Conference on Supercomputing*, pages 44–53, Jul 1994.

[14] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct 1991.